10 points.
Due: Monday, November 29, 2021at 11:59 pm

Section 1 Problem Statement

For assignment 5, we will create a program to simulate buying and selling stocks on a stock market. In particular, this program will manage multiple users, who can each buy/sell various stocks.

Note: The focus of this assignment is on using the Standard Template Library containers and algorithms we covered in the past couple weeks. In the assignment specification, we'll mention different containers and algorithms to use for different features. However, you can treat these as recommendations.
So long as the program behaves correctly, and the Stockholder class functions are correct, you are free to choose where to use STL containers, where to write your own code from scratch, etc. That said, this program has more complicated behavior than previous assignments, so effective use of STL containers/algorithms will be important to completing the program with minimal hassle.
The one exception to this rule is the NetWorth() function for Stockholder, which *must* use the PortfolioValue functor and `for_each` algorithm.

Section 2 Classes

In this section, we'll give details on how to implement the program outlined above.

(a)

Stock

This is a struct, and is given to you as starter code. Each stock contains "short" and "long" names, where the long name is a full company game, and the short name is a 2-4 character abbreviation, as you might see on a stock ticker. A stock also has "old" and "new" values, which are used to represent the stock price at the start and end of a day. Lastly, each stock has a private "bias" value, which is used internally to make a given stock more likely to grow or fall in value.

The public functions for the `Stock` struct are listed below:

- `Stock()` (default constructor)
  This simply sets the names to blank strings, the values to 0, and the bias to a small random number between -5 and 5. The actual data for stocks will be read from a file.
- `Stock& operator=(Stock& that)`
  The assignment operator has been overridden to allow easy copying of one Stock to another. You many not end up needing this, but it is provided in case you write your program to use significant amounts of copying.
- `istream& operator>>(istream& in, Stock& stock)`
  This input operator is used to read a stock from an input stream.
  We expect one Stock per input line, with space-separated values for the long name, short name, old value, and new value, in that order.
- `void AdvanceDay()`
  This function sets the "old' value equal to the previous "new" value, then generates a random change in price to determine the "new new" value. It is intended to simulate the progression of one day on the market.
- `void Print()`
  This function outputs the short name, long name, new value, and difference between new and old values (daily change) to cout. There is no endline from the print function. You may find this function useful when writing the Print function for Stockholder.

(b)

Stockholder Class

This class simulates an individual buyer/seller of stocks on the market. Each Stockholder has a name, a portfolio of stocks owned (with corresponding counts, e.g. 5 Apple stocks, 7 Google stocks, 110 Downtown Dairy Bar stocks...), and a current amount of cash.

For the portfolio, we recommend using a `map<string, pair<Stock*, int>>`. The pair associates a Stock pointer with a corresponding integer count, and the map allows you to easily index the list by a Stock's name. This may save you some time vs. searching a list whenever a stock is bought/sold. The use of Stock pointers allows you to easily access the current value of a stock, without going back to the global list (in this case, we're not dynamically allocating Stocks in Stockholder, merely holding pointers to Stocks that are allocated *somewhere*).
As noted in section 1, you are not required to use these specific containers.

The public functions for `Stockholder` are listed below:

- `Stockholder(string _name)` (param'ed constructor)
  This should set the Stockholder name to `_name`, initialize your data container(s) to empty, and default the cash to 100,000 (we assume all our stock market players are kinda rich).

- `~Stockholder()` (destructor)
  Assuming you do not use dynamically-allocated objects within Stockholder, this is not required. However, if you perform any dynamic allocation within Stockholder for any reason, you should include a destructor to clean up memory.

- `bool Purchase(Stock* stock, int count)`
  This function takes a pointer to a Stock, and a number of stocks to purchase (for our purposes, we'll assume an unlimited number of stocks available on the market). This function should use the "new" value of the Stock to calculate the total price of purchasing `count` shares of the stock. If that price is $<=$ the Stockholder's current cash, then add the Stocks to the Stockholder's portfolio and subtract the cost from their current cash. The function should then return true.
  If the price is $>$ current cash, return false without changing the Stockholder data.

- `bool Sell(string short_name, int count)`
  This function should check the Stockholder's portfolio to see if they have at least `count` shares of the stock with given name.
  If so, multiply the "new" value of the Stock by the count to calculate the total value of the stocks, and add that amount to the Stockholder's current cash. Finally, subtract `count` shares from their portfolio, and return true. If the Stockholder does not hold enough shares, return false.

- `float NetWorth()`
  This function calculates the Stockholder's combined cash and stock value. First, create an instance of the PortfolioValue functor (described below), and use the `for_each` algorithm to calculate the total value of the Stockholder's portfolio. The `for_each` function returns a copy of the PortfolioValue object, with a sum value equal to the total portfolio value. Then return the portfolio value plus the Stockholder's current cash.

- `void Print()`
  This function should output the stockholder's name and NetWorth (in dollars) on one line. Then, for each stock in the portfolio, output the short name, long name, "new" value, daily change, and count.
  You may find it helpful to use the `Print()` function from Stock. Note that the "count" here is the count in the portfolio, and is not part of the Stock class. Each stock should go on its own line in the output.

(c)

PortfolioValue

In order to calculate the total value of a Stockholder's portfolio, you should create a functor (a class/struct that overrides the `()` operator). In this case, you should make a struct called PortfolioValue, whose only member variable is a float called `total_value`.

You should then create the following override:
`void operator()(IteratorType)`, where IteratorType is whatever type of data is iterated over in the container you chose for the portfolio. For example, if you used a `vector<pair<Stock*, int>>`, then the IteratorType is `pair<Stock*, int>`, since each element of the container is a pair. If you used the map container we recommended, then IteratorType is `pair<string, pair<Stock*, int>>`, since each element of the map is a pair mapping from a string to a sub-pair. This operator should multiply the count by the Stock's "new" value, then add this to `total_value`.

Section 3 Main Program

The main program is responsible for managing a collection of Stockholders and Stocks. When the program begins, it should open a file called `stocks.txt`, which we will provide. Like the `menu.txt` file in assignment 4, the first line will give the number of Stocks in the file, and each remaining line will contain data for one Stock. Your program should read in each of these stocks, and store it in a container. You may wish to use a `list<Stock>`, or a `map<string, Stock>`. Using a map would allow you to index by the short name of the Stock, for easy access. Alternately, you may prefer to use a list/vector, and use the `find_if` algorithm to search for a Stock with a given name.

In either case, once the file has been read, the main program should proceed to the main menu. In addition to the container of Stocks, your program should have a container of Stockholders. As with the Stocks, we recommend a `map<string, Stockholder` for easy indexing by Stockholder name.

The menus and menu items for the main program will be as follows:

(a)

Main Menu

- In the main menu, the options are to log in, list directory, advance a day, or exit.
- If the user chooses "log in," they will be prompted for their name. You can assume each user has a single-word name (i.e. no spaces in their name).
  If there is not already a Stockholder with the given name in the collection, insert/emplace a new Stockholder with the given name (note that we're using the word "new" generically; you are not *required* to dynamically allocate your Stocholders). In either case, the program should then go to a user menu, where the user has options to list **all** stocks, list **their** stocks, buy stocks, sell stocks, or log out.
- If the user chooses to "list directory," the program will print the names of all users, and wait for the user to enter a number before printing the main menu again.
- If the user chooses "advance a day" the program should call the `AdvanceDay()` function on each stock in the collection.
- If the user chooses "exit" the program ends. There is no need to write any files out; we won't save Stockholder data across program runs.

(b)

User Menu

- As mentioned above, the user menu should contain options to view all stocks, the user's stocks, to purchase stocks, to sell stocks, or to log out.

- If the user chooses "view all stocks," the program should iterate over the global Stock container, calling Print() on each Stock to display the stock information. It should then wait for the user to enter a number before printing the main menu again. Don't forget print endline characters between Stocks, since the Print() function does not include one.
- If the user choose to "view user stocks," the program should call Print() on the currently logged in Stockholder. It should then wait for the user to enter a number before printing the main menu again.
- If the user chooses "purchase stocks," the program should prompt for the user to enter a Stock's short name. If the user enters a short name that does not match one of the Stocks in the global container, the program should print a short warning message and return to the user menu.
  If the user enters the name for a valid Stock, the program then prompts for the number of shares of that Stock to purchase. You can assume the user will enter a positive number (no need to check for bad inputs here). The program should then call Purchase on the Stockholder, passing in a pointer to the chosen Stock and the chosen count.
  The program should print an appropriate message depending on whether Purchase returned true or false, and return to the user menu.
  (e.g. for true, it may print "Congratulations on purchasing `{count} {s.long_name}` shares!", for false, it might print "Sorry, you can not afford this purchase.")
- Similar to "purchase," if the user chooses "sell stocks," the program should prompt for the user to enter a Stock's short name. If the user does not have that stock in their portfolio, the program should print a short warning message and return to the user menu.
  If the user enters the name for a valid Stock, the program then prompts for the number of shares of that Stock to sell. You can assume the user will enter a positive number (again, no need to check for bad inputs). The program should then call Sell on the Stockholder, passing in the given Stock name, and the count.
  The program should print an appropriate message depending on whether Sell returned true or false, and return to the user menu.
  (e.g. for true, it may print "Congratulations on your sale!", for false, it might print "Sorry, you don't have enough stocks to sell.")

(c)

Menu Formatting/Inputs

Each menu item must have a number indicating the value for the user to enter to pick that menu item (see example menus below).

As with earlier assignments, if the user enters a number that does **not** correspond to a menu entry, your program should re-prompt the user and read in a new number. Again, this "re-prompt" can either re-print the menu, or print a new prompt. The choice is yours, so long as your program does not wait *silently* for valid input. Continue to prompt the user until the number they enter is a valid menu item.

You do not need to worry about the user entering invalid types of data (e.g. they will not enter a character where a number is expected).

Below, we've given examples of each menu, which show what items the menus should have, and in what order they should go. Your program is **not** required to print exact copies of the examples here (though you're welcome to copy the example formats if you like), but you **must** keep the menu items in the same order. While exact text of each menu item is up to you, the items should have reasonable labels.

Here's an example of the main menu:

```
Welcome to the Stock Portfolio Manager!
[1] Log In
[2] List Directory
[3] Advance a Day
[4] Exit program
Please enter a menu item:
```

And an example of the "User" menu:

```
What would you like to do?
[1] List Available Stocks
[2] List My Stocks
[3] Purchase Stocks
[4] Sell Stocks
[5] Log Out (return to main menu)
Please enter a menu item:
```

An example of the "list directory" printout:

```
The following users are available:
Alice
Bob
Charlie
Donna
Eve
Please enter any number to return to the main menu:
```

Example of showing user's stocks (showing "all" stocks is similar, without the user name and counts):

```
Alice $38976.13
NVDA NvidiaCorp $300.25 $5.08 10
WMT WalmartInc $146.91 -$3.14 7
NFLX NetflixInc $679.33 -$2.53 22
Enter any number to return to menu.
```

Section 4 File Example

Below, we'll also show an example of what the stocks.txt file should look like. Note, we'll aslo provide an actual sample stocks.txt.

```
10
TeslaInc        TSLA 1013.39 1013.39
NvidiaCorp      NVDA 300.25 300.25
VisaInc         V    212.30 212.30
JohnsonJohnson JNJ  163.52 163.52
WalmartInc      WMT  146.91 146.91
HomeDepot       HD   371.08 371.08
AdobeInc        ADBE 659.73 659.73
NetflixInc      NFLX 679.33 679.33
WaltDisneyCo    DIS  158.43 158.43
NikeInc         NKE  168.85 168.85
```

Section 5 Submission

Submit a zip file, which contains two .cpp files called `main.cpp` and `Stockholder.cpp`, as well as header files `Stockholder.h` and `Stock.h`.

Your zip file should be named *NetID*`_asg5.zip`, where *NetID* is your Net ID (the username you use on Canvas, *not* the 10-digit number on your wiscard).

In each cpp and header file, other than Stock.h, you should include a comment at the top of the file with your name and NetID.

Section 6 Hints

To build the program, use the following command: `g++ main.cpp Stockholder.cpp -o main`

As stated previously, we've recommended some containers and STL algorithms to use in developing your program. You are, however, free to use whatever STL containers/algos you like, if any.

In order to get unique random values each time you run the program, you should `#include <cstdlib>` and `#include <ctime>` in your main.cpp, then put the line `srand((unsigned int) time(NULL));` at the top of your `main()` function.

Section 7 Rubric

Item	**Points**
Program builds and runs without compile errors	2
The main menu has the correct entries, and correctly lets users log in	1
The program manages a collection of stockholders, and successfully prints all stockholder names when "list directory" is chosen	1
The program loads and maintains a collection of stocks, and successfully advances all stocks when "advance day" is chosen	1
User menu has correct entries, and correctly lets users list all available stocks	1
Stockholder maintains a portfolio of stocks, and changes when advancing the day are reflected in the Stockholder portfolio when Print() is called	1
Stockholder successfully implements the NetWorth function, using a PortfolioValue functor with the `for_each` algorithm	1
Stockholder successfully implements the Purchase and Sell functions	2
Total	**10**